CHAPTER 4

Plotting and Visualization

Visualization is a universal tool for investigating problems and communicating the results of computational studies. It is hardly an exaggeration to say that the end product of nearly all computations—numeric or symbolic—is a plot or a graph. When visualized in a graphical form, knowledge and insights can be easily gained from computational results. Visualization is a tremendously important part of the workflow in all fields of computational studies.

There are many high-quality visualization libraries in the scientific computing environment for Python. The most popular general-purpose visualization library is Matplotlib, which mainly focuses on generating static publication-quality 2D and 3D graphs. Many other libraries focus on niche areas of visualization. A few prominent examples are Bokeh (`http://bokeh.pydata.org`) and Plotly (`http://plot.ly`), which both primarily focus on interactivity and web connectivity, Seaborn (`http://seaborn.pydata.org`) which is a high-level plotting library that targets statistical data analysis and which is based on the Matplotlib library, and the Mayavi library (`http://docs.enthought.com/mayavi/mayavi`) for high-quality 3D visualization, which uses the venerable VTK software (`http://www.vtk.org`) for heavy-duty scientific visualization. It is also worth noting that other VTK-based visualization software, such as ParaView (`http://www.paraview.org`), is scriptable with Python and can also be controlled from Python applications. In the 3D visualization space, there are also more recent players, such as VisPy (`http://vispy.org`), an OpenGL-based 2D and 3D visualization library with great interactivity and connectivity with browser-based environments, such as the Jupyter Notebook.

The visualization landscape in the scientific computing environment for Python is vibrant and diverse, providing ample options for various visualization needs. This chapter explores traditional scientific visualization in Python using the Matplotlib library, including plots and figures commonly used to visualize results and data in scientific and technical disciplines, such as line plots, bar plots, contour plots, colormap plots, and 3D surface plots.

Matplotlib Matplotlib is a Python library for publication-quality 2D and 3D graphics, supporting various output formats. At the time of writing, the latest version is 3.7.2. More information about Matplotlib is available at `www.matplotlib.org`. This website contains detailed documentation and an extensive gallery showcasing the various graphs that can be generated using the Matplotlib library, together with the code for each example. This gallery is a great source of inspiration for visualization ideas, and I highly recommend exploring Matplotlib by browsing this gallery.

There are two common approaches to creating scientific visualizations: using a graphical user interface to manually build up graphs and using a programmatic approach where the graphs are created with code. Both approaches have their advantages and disadvantages. This chapter takes the programmatic approach and explores how to use the Matplotlib API to create graphs and control every aspect of their appearance. The programmatic approach is a particularly suitable method for creating graphics for scientific and technical applications, particularly for creating publication-quality figures. An important part of the motivation for this is that programmatically created graphics can guarantee consistency across multiple figures, make reproducible, and easily be revised and adjusted without having to redo potentially lengthy and tedious procedures in a graphical user interface.

Importing Modules

Unlike most Python libraries, Matplotlib provides multiple entry points into the library with different application programming interfaces (APIs). Specifically, it provides a stateful API and an object-oriented API, both provided by the `matplotlib.pyplot` module. I strongly recommend only using the object-oriented approach, and the remainder of this chapter solely focuses on this part of Matplotlib.[1]

To use the object-oriented Matplotlib API, we first need to import its Python modules. The following assumes that Matplotlib is imported using the following standard convention.

```
In [1]: %matplotlib inline
In [2]: import matplotlib as mpl
In [3]: import matplotlib.pyplot as plt
In [4]: from mpl_toolkits.mplot3d.axes3d import Axes3D
```

The first line assumes that we are working in an IPython environment, specifically in the Jupyter Notebook or the IPython QtConsole. The IPython magic command `%matplotlib inline` configures the Matplotlib to use the "inline" backend, which results in the created figures being displayed directly in, for example, the Jupyter Notebook rather than in a new window. The statement `import matplotlib as mpl` imports the main Matplotlib module, and the import statement `import matplotlib.pyplot as plt`, is for convenient access to the submodule `matplotlib.pyplot` that provides the functions used to create new `Figure` instances.

This chapter makes frequent use of the NumPy library and, as in Chapter 2, assumes that NumPy is imported using the following.

```
In [5]: import numpy as np
```

The SymPy library is also used, imported as follows.

```
In [6]: import sympy
```

Getting Started

Before delving deeper into creating graphics with Matplotlib, let's begin with an example of creating a simple but typical graph. Then, let's cover some of the fundamental principles of the Matplotlib library, to build up an understanding of how graphics can be produced with the library.

A graph in Matplotlib is structured in terms of a `Figure` instance and one or more `Axes` instances within the figure. The `Figure` instance provides a canvas area for drawing, and the `Axes` instances provide coordinate systems assigned to fixed regions of the total figure canvas; see Figure 4-1.

[1] Although the stateful API may be convenient and simple for small examples, the readability and maintainability of code written for stateful APIs scale poorly, and the context-dependent nature of such code makes it hard to rearrange or reuse. I therefore recommend to avoid it altogether and to only use the object-oriented API.

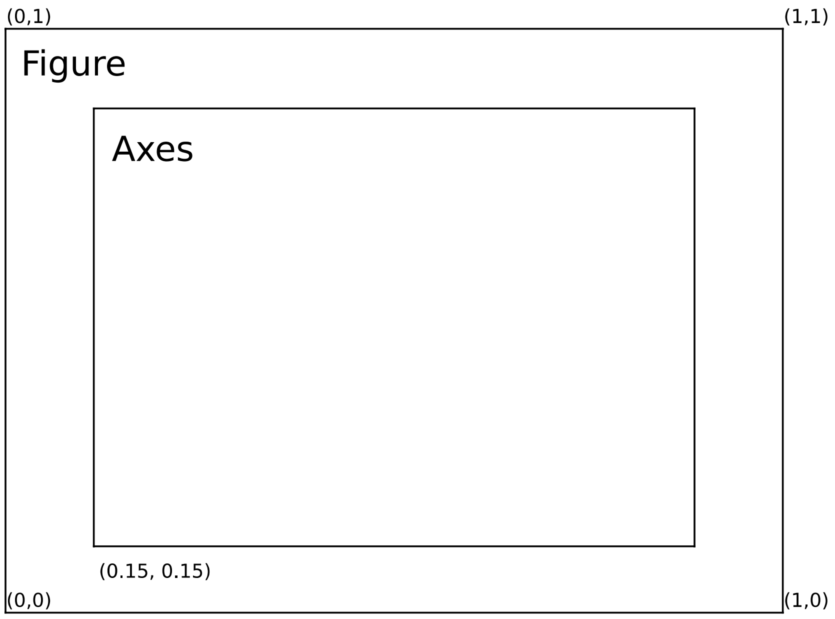


Figure 4-1. *Illustration of the arrangement of a Matplotlib `Figure` instance and an `Axes` instance. The `Axes` instance provides a coordinate system for plotting, and it is assigned to a region within the figure canvas. The figure canvas has a simple coordinate system where (0, 0) is the lower-left corner and (1,1) is the upper-right corner. This coordinate system is only used when placing elements, such as `Axes`, directly on the figure canvas*

A `Figure` can contain multiple `Axes` instances, for example, to show multiple panels in a figure or to show insets within another `Axes` instance. An `Axes` instance can manually be assigned to an arbitrary region of a figure canvas, or `Axes` instances can be automatically added to a figure canvas using one of several layout managers provided by Matplotlib. The `Axes` instance provides a coordinate system that can be used to plot data in various plot styles, including line graphs, scatter plots, bar plots, and many others. In addition, the `Axes` instance also determines how the coordinate axes are displayed, for example, with respect to the axis labels, ticks and tick labels, and so on. In fact, when working with Matplotlib's object-oriented API, most functions needed to tune a graph's appearance are methods of the `Axes` class.

As a simple example for getting started with Matplotlib, let's say that we would like to graph the $y(x) = x^3+5x^2+10$ function, together with its first- and second-order derivatives, over the range $x \in [-5, 2]$. To do this, we first create NumPy arrays for the x range and then compute the three functions we want to graph. When the data for the graph is prepared, we need to create Matplotlib `Figure` and `Axes` instances and then use the `plot` method of the `Axes` instance to plot the data. We can set basic graph properties such as x and y-axis labels using the `set_xlabel` and `set_ylabel` methods and generate a legend using the `legend` method. These steps are carried out in the following code, and the resulting graph is shown in Figure 4-2.

```
In [7]: x = np.linspace(-5, 2, 100)
   ...: y1 = x**3 + 5*x**2 + 10
   ...: y2 = 3*x**2 + 10*x
   ...: y3 = 6*x + 10
   ...:
   ...: fig, ax = plt.subplots()
   ...: ax.plot(x, y1, color="blue", label="y(x)")
   ...: ax.plot(x, y2, color="red", label="y'(x)")
   ...: ax.plot(x, y3, color="green", label="y”(x)")
   ...: ax.set_xlabel("x")
   ...: ax.set_ylabel("y")
   ...: ax.legend()
```

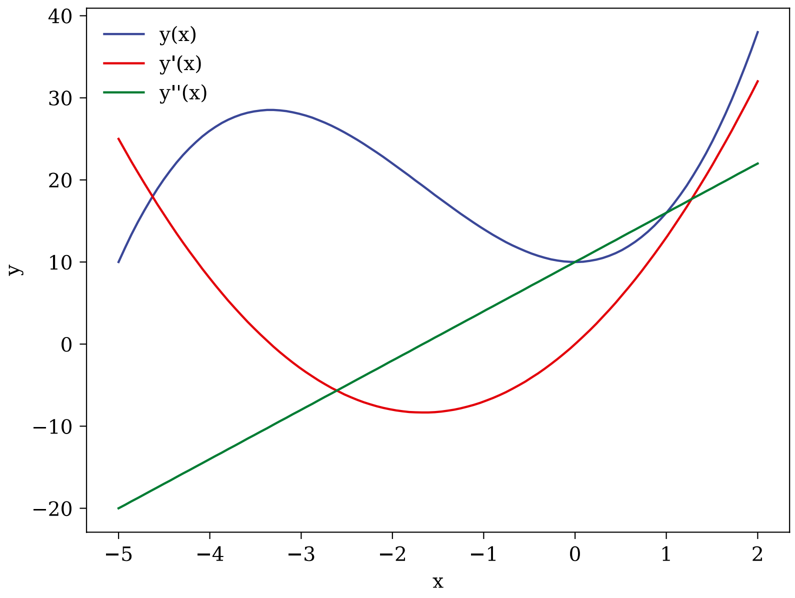


Figure 4-2. *Example of a simple graph created with Matplotlib*

The `plt.subplots` function generated the `Figure` and `Axes` instances. This function can be used to create grids of `Axes` instances within a newly created `Figure` instance, but here, it was merely used as a convenient way of creating a `Figure` and an `Axes` instance in one function call. Once the `Axes` instance is available, note that all the remaining steps involve the calling methods of this `Axes` instance. To graph the data, we use `ax.plot`, which takes as first and second arguments NumPy arrays with numerical data for the *x* and *y* values of the graph, and it draws a line connecting these data points. We also used the optional `color` and `label` keyword arguments to specify the color and assign a text label to each line used in the legend. These few lines of code are enough to generate the graph we set out to produce, but as a bare minimum, we should also set labels on the *x* and *y* axes and, if suitable, add a legend for the curves we have plotted. The axis labels are set with `ax.set_xlabel` and `ax.set_ylabel` methods, which take a text string with the corresponding label as an argument. The legend is added using the `ax.legend` method, which does not require any arguments since the `label` keyword argument was used when plotting the curves.

These are the typical steps required to create a graph using Matplotlib. While the graph in Figure 4-2 is complete and fully functional, there is certainly room for improvements in many aspects of its appearance. For example, to meet publication or production standards, we may need to change the font and the font size of the axis labels, the tick labels, and the legend, and we should probably move the legend to a part of the graph where it does not interfere with the curves we are plotting. We might even want to change the number of axis ticks and labels and add annotations and helplines to emphasize certain aspects of the graph. With a few changes along these lines, the figure may, for example, appear like in Figure 4-3, which is considerably more presentable. The remainder of this chapter examines how to fully control the appearance of the graphics produced using Matplotlib.

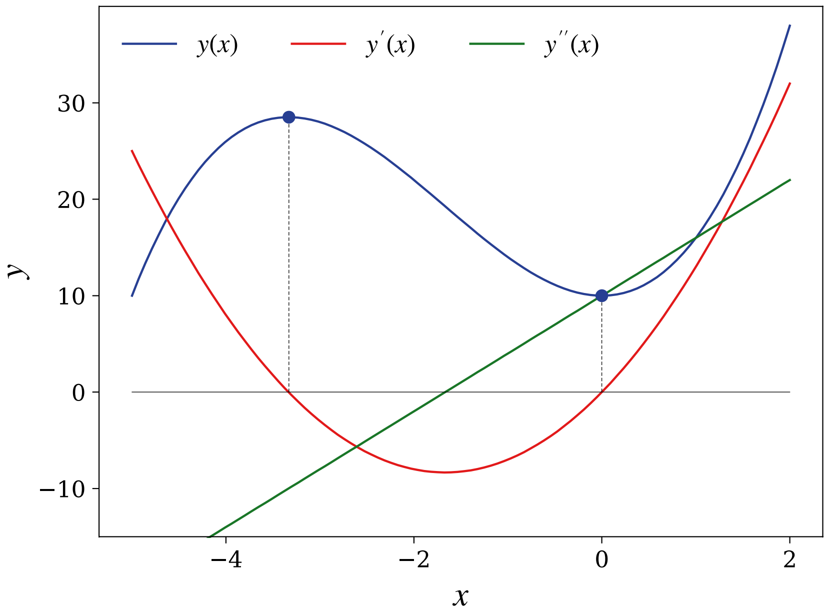


Figure 4-3. *Revised version of Figure 4-2*

Interactive and Noninteractive Modes

The Matplotlib library is designed to work well with many different environments and platforms. As such, the library not only contains routines for generating graphs but also supports displaying graphs in different graphical environments. To this end, Matplotlib provides *backends* for generating graphics in different formats (e.g., PNG, PDF, Postscript, and SVG) and for displaying graphics in a graphical user interface using a variety of different widget toolkits (e.g., Qt, GTK, wxWidgets, and Cocoa for macOS) that are suitable for different platforms.

The backend can be selected in that Matplotlib resource file[2] or using the `mpl.use` function, which must be called right after importing `matplotlib`, before importing the `matplotlib.pyplot` module. For example, to select the Qt5Agg backend, we can use the following.

```
import matplotlib as mpl
mpl.use('Qt5Agg')
import matplotlib.pyplot as plt
```

[2] The Matplotlib resource file, `matplotlibrc`, can be used to set default values of many Matplotlib parameters, including which backend to use. The location of the file is platform dependent. For details, see `http://matplotlib.org/stable/tutorials/introductory/customizing.html`.

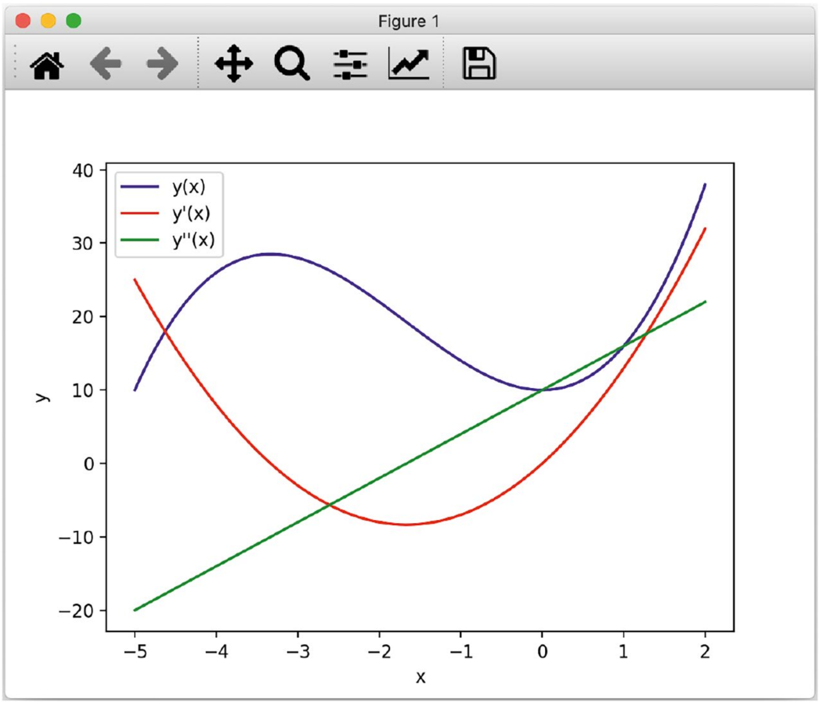


Figure 4-4. *A screenshot of the Matplotlib graphical user interface for displaying figures, using the Qt5 backend on macOS. The detailed appearance varies across platforms and backends, but the basic functionality is the same*

The graphical user interface for displaying Matplotlib figures, as shown in Figure 4-4, is useful for interactive use with Python script files or the IPython console, and it allows us to interactively explore figures, for example, by zooming and panning. When using an interactive backend, which displays the figure in a graphical user interface, it is necessary to call the `plt.show` function to get the window to appear on the screen. By default, the `plt.show` function call hangs until the window is closed. For a more interactive experience, we can activate *interactive mode* by calling the `plt.ion` function. This instructs Matplotlib to take over the GUI event loop and show a window for a figure as soon as it is created, returning the control flow to the Python or IPython interpreter. To have the changes to a figure take effect, we must issue a redraw command using the `plt.draw` function. We can deactivate the interactive mode using the `plt.ioff` function, and we can use the `mpl.is_interactive` function to check if Matplotlib is in interactive or noninteractive mode.While the interactive graphical user interfaces have unique advantages, when working the Jupyter Notebook or the Qtconsole, it is often more convenient to display Matplotlib-produced graphics embedded directly in the notebook. This behavior is the default in the current version of Jupyter, but the “inline backend” can also be activated explicitly using the IPython command `%matplotlib inline`. This configures Matplotlib to use a noninteractive backend to generate graphics images, which are then displayed as static images in, for example, the Jupyter Notebook. The inline backend for Matplotlib can be fine-tuned using the IPython `%config` command. For example, we can select the output format for the generated graphics using the `InlineBackend.figure_format` option,[3] which, for example, we can set to ‘svg’ to generate SVG graphics rather than PNG files.

[3] For macOS users, `%config InlineBackend.figure_format='retina'` is another useful option, which improves the quality of the Matplotlib graphics when viewed on retina displays.

```
In [8]: %matplotlib inline
In [9]: %config InlineBackend.figure_format='svg'
```

With this approach, the interactive aspect of the graphical user interface is lost (e.g., zooming and panning), but embedding the graphics directly in the notebook has many other advantages. For example, keeping the code that was used to generate a figure together with the resulting figure in the same document eliminates the need for rerunning the code to display a figure, and the interactive nature of the Jupyter Notebook itself replaces some of the interactivity of Matplotlib's graphical user interface.

When using the IPython inline backend, it is unnecessary to use `plt.show` and `plt.draw`, since the IPython rich display system is responsible for triggering the rendering and displaying of the figures. This book assumes that code examples are executed in the Jupyter Notebooks, and the calls to the `plt.show` function are not in the code examples. When using an interactive backend, adding this function call at the end of each example is necessary.

Figure

As introduced in the previous section, the `Figure` object is used in Matplotlib to represent a graph. In addition to providing a canvas to place `Axes` instances on, the `Figure` object also provides methods for performing actions on figures, and it has several attributes that can be used to configure the properties of a figure.

A `Figure` object can be created using the `plt.figure` function, which takes several optional keyword arguments for setting figure properties. Notably, it accepts the `figsize` keyword argument, which should be assigned to a tuple on the form `(width, height)`, specifying the width and height of the figure canvas in inches. It can also be useful to specify the color of the figure canvas by setting the `facecolor` keyword argument.

Once a `Figure` is created, we can use the `add_axes` method to create a new `Axes` instance and assign it to a region on the figure canvas. The `add_axes` method takes one mandatory argument: a list containing the coordinates of the lower-left corner and the width and height of the `Axes` in the figure canvas coordinate system in the `(left, bottom, width, height)` format.[4] The coordinates and the width and height of the `Axes` object are expressed as fractions of the total canvas width and height; see Figure 4-1. For example, an `Axes` object that completely fills the canvas corresponds to `(0, 0, 1, 1)`, leaving no space for axis labels and ticks. A more practical size could be `(0.1, 0.1, 0.8, 0.8)`, corresponding to a centered `Axes` instance that covers 80% of the width and height of the canvas. The `add_axes` method takes a large number of keyword arguments for setting properties of the new `Axes` instance. These are described in more detail later in this chapter when I discuss the `Axes` object in depth. However, one keyword argument worth emphasizing here is `facecolor`, with which we can assign a background color for the `Axes` object. Together with the `facecolor` argument of `plt.figure`, this allows selecting colors of both the canvas and the regions covered by `Axes` instances.

With the `Figure` and `Axes` objects obtained from `plt.figure` and `fig.add_axes`, we have the necessary preparations to start plotting data using the methods of the `Axes` objects. See the next section of this chapter for more details on this. However, once the required plots have been created, more methods in the Figure objects are important in the graph-building workflow. For example, to set an overall figure title, we can use `suptitle`, which takes a title as a string as an argument. To save a figure to a file, we can use the `savefig` method. This method takes a string with the output filename as the first argument and several optional keyword arguments. By default, the output file format is determined by the file extension of the filename argument, but we can also specify the format explicitly using the `format` argument. The available output formats depend on which Matplotlib backend is used, but commonly available options are PNG, PDF, EPS,

[4] An alternative to passing a coordinate and size tuple to `add_axes` is to pass an existing Axes instance to add it to the figure.

and SVG formats. The resolution of the generated image can be set with the `dpi` argument. DPI stands for "dots per inch," and since the figure size is specified in inches using the `figsize` argument, multiplying these numbers gives the output image size in pixels. For example, with `figsize=(8, 6)` and `dpi=100`, the size of the generated image is 800x600 pixels. The `savefig` method also takes some arguments similar to those of the `plt.figure` function, such as the `facecolor` argument. Finally, the figure canvas can be made transparent using the `transparent=True` argument to `savefig`. The following code listing illustrates these techniques; the result is shown in Figure 4-5.

```
In [10]: fig = plt.figure(figsize=(8, 2.5), facecolor="#f1f1f1")
    ...:
    ...: # axes coordinates as fractions of the canvas width and height
    ...: left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
    ...: ax = fig.add_axes((left, bottom, width, height), facecolor="#e1e1e1")
    ...:
    ...: x = np.linspace(-2, 2, 1000)
    ...: y1 = np.cos(40 * x)
    ...: y2 = np.exp(-x**2)
    ...:
    ...: ax.plot(x, y1 * y2)
    ...: ax.plot(x, y2, 'g')
    ...: ax.plot(x, -y2, 'g')
    ...: ax.set_xlabel("x")
    ...: ax.set_ylabel("y")
    ...:
    ...: fig.savefig("graph.png", dpi=100)
```

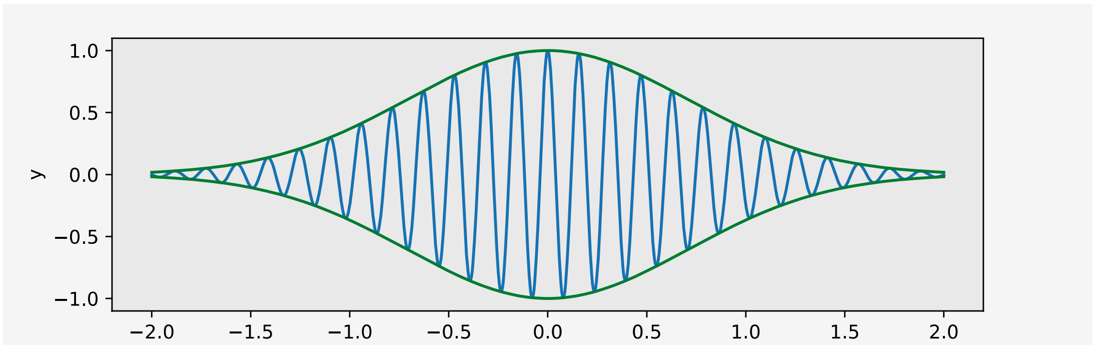


Figure 4-5. *Graph showing the result of setting the size of a figure with `figsize`, adding a new `Axes` instance with `add_axes`, setting the background colors of the `Figure` and `Axes` objects using `facecolor`, and finally saving the figure to file using `savefig`*

Axes

The `Figure` object introduced in the previous section provides the backbone of a Matplotlib graph, but all the interesting content is organized within or around `Axes` instances. We have already encountered `Axes` objects in this chapter. The `Axes` object is central to most plotting activities with the Matplotlib library. It provides a coordinate system to plot data and mathematical functions. In addition, it contains the axis objects that determine where the axis labels and the axis ticks are placed. The functions for drawing different types of plots are also methods of this Axes class. This section first explores different types of plots that can be drawn using Axes methods and how to customize the appearance of the *x* and *y* axes and the coordinate systems used with an Axes object.

We have seen how new Axes instances can be added to a figure explicitly using the add_axes method. This is a flexible and powerful method for placing Axes objects at arbitrary positions, which has several important applications, as explained later in the chapter. However, for most common uses, it is tedious to specify explicitly the coordinates of the Axes instances within the figure canvas. This is especially true when using multiple panels of Axes instances within a figure, for example, in a grid layout. Matplotlib provides several Axes layout managers, creating and placing Axes instances within a figure canvas following different strategies. Later in this chapter, we learn about using such layout managers. However, to facilitate the forthcoming examples, we briefly look at one of these layout managers: the plt.subplots function. Earlier in this chapter, this function conveniently generated new Figure and Axes objects in one function call. However, the plt.subplots function is also capable of filling a figure with a grid of Axes instances, which is specified using the first and the second arguments or with the nrows and ncols arguments, which, as the names imply, create a grid of Axes objects, with the given number of rows and columns. For example, to generate a grid of Axes instances in a newly created Figure object with three rows and two columns, we can use the following.

```
fig, axes = plt.subplots(nrows=3, ncols=2)
```

Here, the plt.subplots function returns a tuple (fig, axes), where fig is a Figure instance, and axes is a NumPy array of size (nrows, ncols), in which each element is an Axes instance that has been appropriately placed in the corresponding figure canvas. At this point, we can also specify that columns and/or rows should share the *x* and *y* axes using the sharex and sharey arguments, which can be set to True or False.

The plt.subplots function also takes two special keyword arguments fig_kw and subplot_kw, which are dictionaries with keyword arguments used when creating the Figure and Axes instances, respectively. This allows setting the properties of the Figure and Axes objects with plt.subplots similarly to when directly using plt.figure and the make_axes method.

Plot Types

Effective scientific and technical data visualization requires a wide variety of graphing techniques. Matplotlib implements many types of plotting techniques as methods of the Axes object. For example, the previous examples used the plot method, which draws curves in the coordinate system provided by the Axes object. The following sections explore some of Matplotlib's plotting functions in more depth by using these functions in example graphs. A summary of commonly used 2D plot functions is shown in Figure 4-6. Other types of graphs, such as color maps and 3D graphs, are discussed later in this chapter. All plotting functions in Matplotlib expect data as NumPy arrays as input, typically arrays with *x* and *y* coordinates as the first and second arguments. For details, see the docstrings for each method shown in Figure 4-6, using, for example, help(plt.Axes.bar).

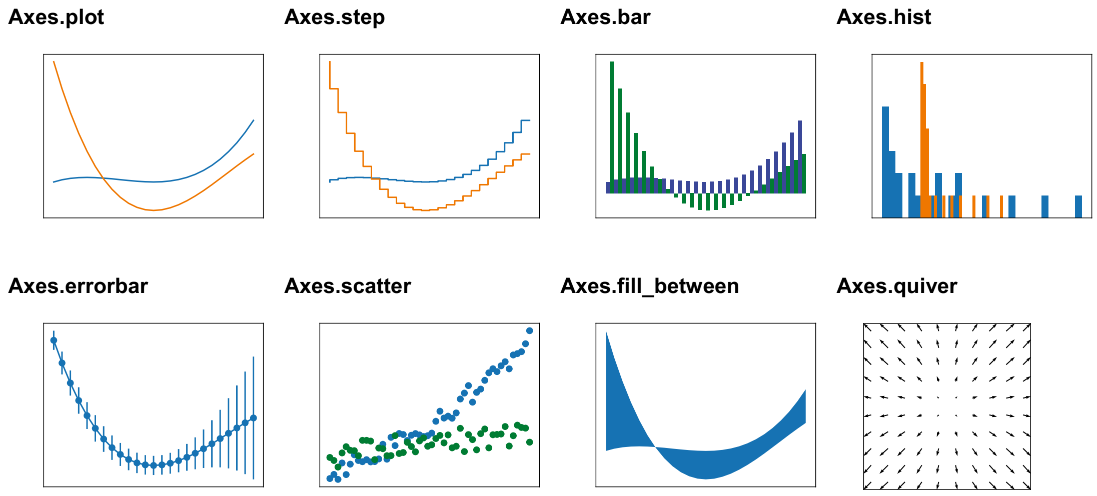


Figure 4-6. *Overview of selected 2D graph types. The name of the* `Axes` *method for generating each type of graph is shown together with the corresponding graph*

Line Properties

The most basic type of plot is the simple line plot. It may, for example, be used to depict the graph of a univariate function or to plot data as a function of a control variable. In line plots, we frequently need to configure the properties of the lines in the graph, for example, the line width, line color, and line style (solid, dashed, dotted, etc.). In Matplotlib we set these properties with keyword arguments to the plot methods, such as `plot`, `step`, and `bar`. A few of these graph types are shown in Figure 4-6. Many plot methods have specific arguments, but basic properties such as colors and line width are shared among most plotting methods. These properties and the corresponding keyword arguments are summarized in Table 4-1.

Table 4-1. *Basic Line Properties and Their Corresponding Argument Names for Use with the Matplotlib Plotting Methods*

Argument	Example Values	Description
`color`	A color specification can be a string with a color name, such as "red," "blue," or an RGB color code on the form "#aabbcc"	A color specification
`alpha`	Float number between 00 (completely transparent) and 1.0 (completely opaque)	The amount of transparency
`linewidth, lw`	Float number	The width of a line
`linestyle, ls`	"-" – solid "--" – dashed ":" – dotted ".-" – dash-dotted	The style of the line (i.e., whether the line is to be drawn as a solid line or if it should be, for example, dotted or dashed)
`marker`	+, o, * = cross, circle, star s = square . = small dot 1, 2, 3, 4, ... = triangle-shaped symbols with different angles	Whether or not connected to adjacent data points, each data point can be represented with a marker symbol as specified with this argument
`markersize`	Float number	The marker size
`markerfacecolor`	Color specification	The fill color for the marker
`markeredgewidth`	Float number	The line width of the marker edge
`markeredgecolor`	Color specification	The marker edge color

To illustrate these properties and arguments, consider the following code, which draws horizontal lines with various line width values, line style, marker symbol, color, and size. The resulting graph is shown in Figure 4-7.

```
In [11]: x = np.linspace(-5, 5, 5)
    ...: y = np.ones_like(x)
    ...:
    ...: def axes_settings(fig, ax, title, ymax):
    ...:     ax.set_xticks([])
    ...:     ax.set_yticks([])
    ...:     ax.set_ylim(0, ymax+1)
    ...:     ax.set_title(title)
    ...:
    ...: fig, axes = plt.subplots(1, 4, figsize=(16,3))
    ...:
    ...: # Line width
    ...: linewidths = [0.5, 1.0, 2.0, 4.0]
    ...: for n, linewidth in enumerate(linewidths):
    ...:     axes[0].plot(x, y + n, color="blue", linewidth=linewidth)
    ...: axes_settings(fig, axes[0], "linewidth", len(linewidths))
    ...:
```

```
    ...: # Line style
    ...: linestyles = ['-', '-.', ':']
    ...: for n, linestyle in enumerate(linestyles):
    ...:     axes[1].plot(x, y + n, color="blue", lw=2, linestyle=linestyle)
    ...:
    ...: # custom dash style
    ...: line, = axes[1].plot(x, y + 3, color="blue", lw=2)
    ...: length1, gap1, length2, gap2 = 10, 7, 20, 7
    ...: line.set_dashes([length1, gap1, length2, gap2])
    ...: axes_settings(fig, axes[1], "linetypes", len(linestyles) + 1)
    ...:
    ...: # marker types
    ...: markers = ['+', 'o', '*', 's', '.', '1', '2', '3', '4']
    ...: for n, marker in enumerate(markers):
    ...:     # lw = shorthand for linewidth, ls = shorthand for linestyle
    ...:     axes[2].plot(x, y + n, color="blue", lw=2, ls='None', marker=marker)
    ...: axes_settings(fig, axes[2], "markers", len(markers))
    ...:
    ...: # marker size and color
    ...: markersizecolors = [(4, "white"), (8, "red"), (12, "yellow"),
    ...:                     (16, "lightgreen")]
    ...: for n, (markersize, markerfacecolor) in enumerate (markersizecolors):
    ...:     axes[3].plot(x, y + n, color="blue", lw=1, ls='-',
    ...:                  marker='o', markersize=markersize,
    ...:                  markerfacecolor=markerfacecolor, markeredgewidth=2)
    ...: axes_settings(fig, axes[3], "marker size/color", len (markersizecolors))
```

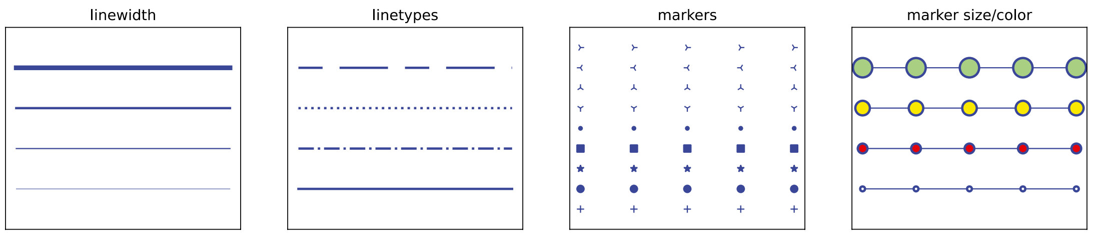


Figure 4-7. *Graphs showing the result of setting the properties line width, line style, marker type, and marker size, and color*

In practice, using different colors, line widths, and line styles are important tools for making a graph easily readable. In a graph with many lines, we can use a combination of colors and style to make each line uniquely identifiable, for example, via a legend. The line width property is best used to emphasize important lines. Consider the following example, where the sin(x) function is plotted with its first few series expansions around $x = 0$, as shown in Figure 4-8.

```
In [12]: # a symbolic variable for x,
    ...: # and a numerical array with specific values of x
    ...: sym_x = sympy.Symbol("x")
    ...: x = np.linspace(-2 * np.pi, 2 * np.pi, 100)
    ...:
    ...: def sin_expansion(x, n):
    ...:     """
```

```
...:     Evaluate the nth order Taylor series expansion
...:     of sin(x) for the numerical values in the array x.
...:     """
...:      return sympy.lambdify(sym_x,
...:                            sympy.sin(sym_x).series(n=n+1).removeO(),
...:                            'numpy')(x)
...:
...: fig, ax = plt.subplots()
...:
...: ax.plot(x, np.sin(x), linewidth=4, color="red", label='exact')
...:
...: colors = ["blue", "black"]
...: linestyles = [':', '-.', '--']
...: for idx, n in enumerate(range(1, 12, 2)):
...:     ax.plot(x, sin_expansion(x, n), color=colors[idx // 3],
...:             linestyle=linestyles[idx % 3], linewidth=3,
...:             label="order %d approx." % (n+1))
...:
...: ax.set_ylim(-1.1, 1.1)
...: ax.set_xlim(-1.5*np.pi, 1.5*np.pi)
...:
...: # place a legend outsize of the Axes
...: ax.legend(bbox_to_anchor=(1.02, 1), loc=2, borderaxespad=0.0)
...: # make room for the legend to the right of the Axes
...: fig.subplots_adjust(right=.75)
```

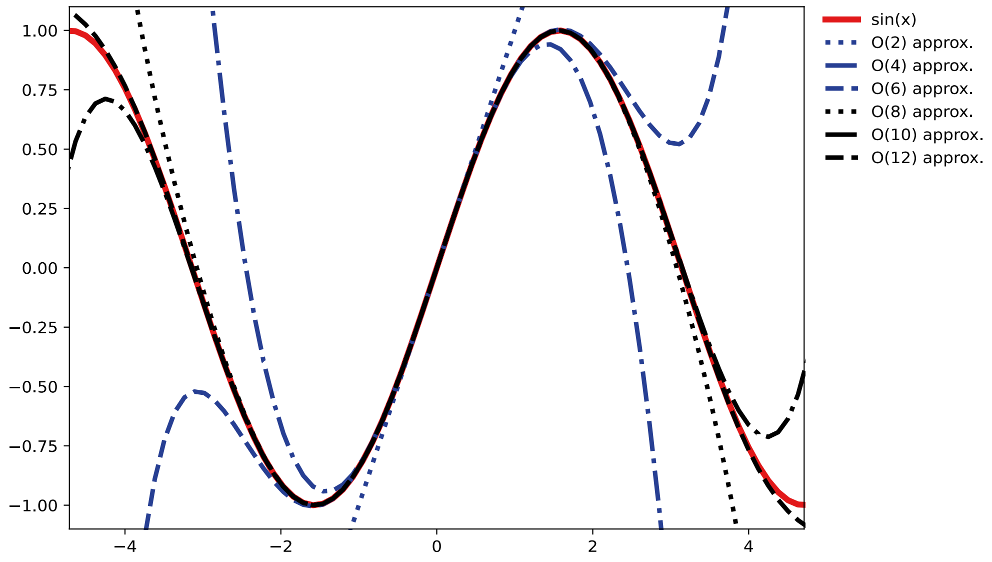


Figure 4-8. *Graph for sin(x) together with its Taylor series approximation of the few lowest orders*

Legends

A graph with multiple lines may often benefit from a legend, which displays a label along each line type somewhere within the figure. As shown in the previous example, using the legend method, a legend may be added to an Axes instance in a Matplotlib figure. Only lines with assigned labels are included in the legend (to assign a label to a line, use the `label` argument of, for example, `Axes.plot`). The legend method accepts a large number of optional arguments. See `help(plt.legend)` for details. Let's emphasize a few of the more useful arguments. The example in the previous section used the `loc` argument, which allows specifying where in the `Axes` area the legend is to be added: `loc=1` for upper-right corner, `loc=2` for upper-left corner, `loc=3` for the lower-left corner, and `loc=4` for the lower-right corner, as shown in Figure 4-9.

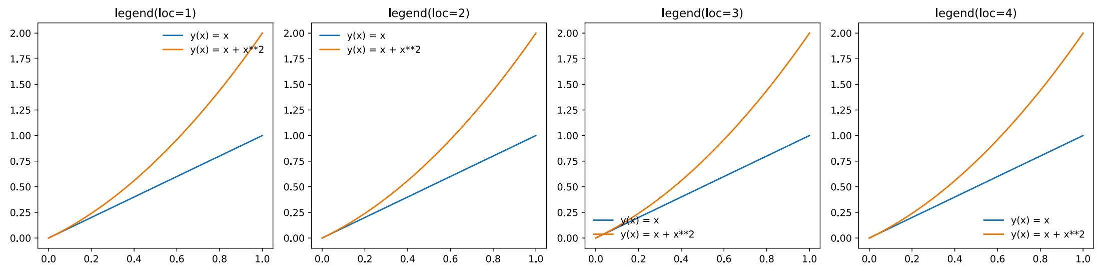


Figure 4-9. *Legend at different positions within an `Axes` instance specified using the `loc` argument of the `legend` method*

The example in the previous section also used the `bbox_to_anchor` argument, which helps place the legend at an arbitrary location within the figure canvas. The `bbox_to_anchor` argument takes the value of a tuple on the form `(x, y)`, where `x` and `y` are the *canvas coordinates* within the `Axes` object. The point `(0, 0)` corresponds to the lower-left corner, and `(1, 1)` corresponds to the upper-right corner. Note that `x` and `y` can be smaller than 0 and larger than 1 in this case, which indicates that the legend is to be placed outside the `Axes` area, like in the previous section.

By default, all lines in the legend are shown in a vertical arrangement. Using the `ncols` argument, it is possible to split the legend labels into multiple columns, as illustrated in Figure 4-10.

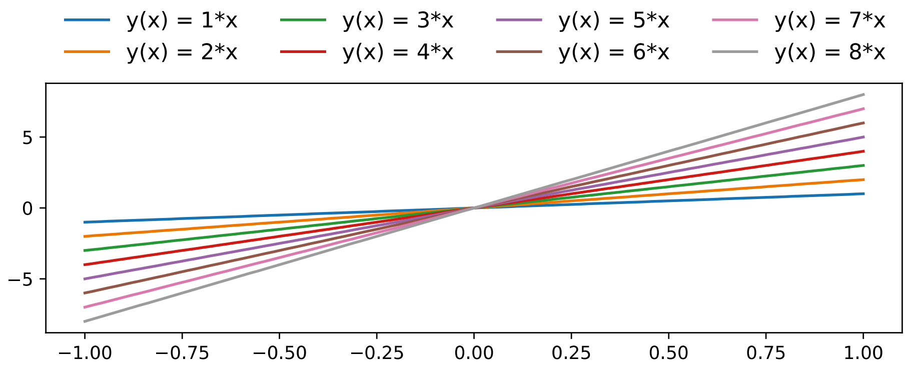


Figure 4-10. *Legend displayed outside the `Axes` object and shown with four columns instead of the single one, using `ax.legend(ncol=4, loc=3, bbox_to_anchor=(0, 1))`*

Text Formatting and Annotations

Text labels, titles, and annotations are important components in most graphs, and having full control of, for example, the font types and font sizes that are used to render such texts is necessary for producing publication-quality graphs. Matplotlib provides several methods to configure font properties. The default values can, for instance, be set in the Matplotlib resource file, and session-wide configuration can be set in the `mpl.rcParams` dictionary. This dictionary is a cache of the Matplotlib resource file and changes to parameters within this dictionary are valid until the Python interpreter is restarted and Matplotlib is imported again. Parameters relevant to how text is displayed include, for example, `'font.family'` and `'font.size'`.

■ **Tip** Use `print(mpl.rcParams)` to get a list of possible configuration parameters and their current values. Updating a parameter is as simple as assigning a new value to the corresponding item in the dictionary `mpl.rcParams`, for example, `mpl.rcParams['savefig.dpi'] = 100`. See also the `mpl.rc` function, which can be used to update the `mpl.rcParams` dictionary, and `mpl.rcdefaults` for restoring the default values.

It is also possible to set text properties on a case-to-case basis by passing a set of standard keyword arguments to functions that create text labels in a graph. Most Matplotlib functions that deal with text labels, in one way or another, accept the keyword arguments summarized in Table 4-2 (this list is an incomplete selection of common arguments; see `help(mpl.text.Text)` for a complete reference). For example, these arguments can be used with the `Axes.text` method, which creates a new text label at a given coordinate. They may also be used with `set_title`, `set_xlabel`, `set_ylabel`, and so on. For more information on these methods, see the next section.

Table 4-2. *Summary of Selected Font Properties and the Corresponding Keyword Arguments*

Argument	Description
`Fontsize`	The size of the font (in points)
`family or fontname`	The font type
`Backgroundcolor`	Color specification for the background color of the text label
`Color`	Color specification for the font color
`Alpha`	Transparency of the font color
`Rotation`	Rotation angle of the text label

In scientific and technical visualization, it is important to be able to render mathematical symbols and expressions in text labels. Matplotlib provides excellent support for this through LaTeX markup within its text labels: any text label in Matplotlib can include LaTeX math expressions by enclosing it within $ signs, for example, `"Regular text: $f(x)=1-x^2$"`. By default, Matplotlib uses an internal LaTeX rendering, which supports a subset of the LaTeX language. However, by setting the configuration parameter `mpl.rcParams["text.usetex"]=True`, it is also possible to use an external full-featured LaTeX engine (if it is available on your system).

When embedding LaTeX code in strings in Python, there is a common stumbling block: Python uses \ as the escape character, whereas in LaTeX, it is used to denote the start of commands. To prevent the Python interpreter from escaping characters in strings containing LaTeX expressions, it is convenient to use raw strings, which are literal string expressions that are prepended with an `r`, for example, `r"$\int f(x) dx$"` and `r'$x_{\rm A}$'`.

The following example demonstrates how to add text labels and annotations to a Matplotlib figure using `ax.text` and `ax.annotate` and how to render a text label that includes an equation typeset in LaTeX. The resulting graph is shown in Figure 4-11.

```
In [13]: fig, ax = plt.subplots(figsize=(12, 3))
    ...:
    ...: ax.set_yticks([])
    ...: ax.set_xticks([])
    ...: ax.set_xlim(-0.5, 3.5)
    ...: ax.set_ylim(-0.05, 0.25)
    ...: ax.axhline(0)
    ...:
    ...: # text label
    ...: ax.text(0, 0.1, "Text label", fontsize=14, family="serif")
    ...:
    ...: # annotation
    ...: ax.plot(1, 0, "o")
    ...: ax.annotate("Annotation",
    ...:             fontsize=14, family="serif",
    ...:             xy=(1, 0), xycoords="data",
    ...:             xytext=(+20, +50), textcoords="offset points",
    ...:             arrowprops=dict(arrowstyle="->",
    ...:                             connectionstyle="arc3,rad=.5"))
    ...:
    ...: # equation
    ...: ax.text(2, 0.1, r"Equation: $i\hbar\partial_t \Psi = \hat{H}\Psi$",
    ...:            fontsize=14, family="serif")
```

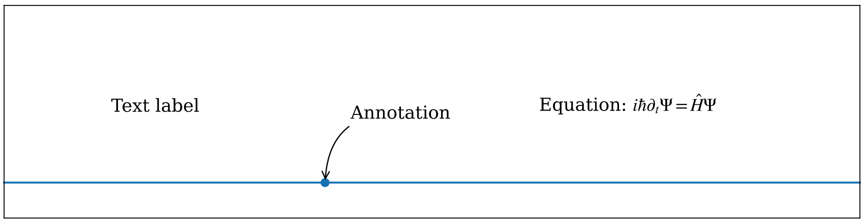


Figure 4-11. *Example demonstrating the result of adding text labels and annotations using `ax.text` and `ax.annotation` and including LaTeX formatted equations in a Matplotlib text label*

Axis Properties

After having created `Figure` and `Axes` objects, the data or functions are plotted using some of the many plot functions provided by Matplotlib, and the appearance of lines and markers are customized—the last major aspect of a graph that remains to be configured and fine-tuned is the Axis instances. A two-dimensional graph has two axis objects: the horizontal *x axis* and the vertical *y axis*. Each axis can be individually configured with respect to attributes such as the axis labels, the placement of ticks and the tick labels, and the location and appearance of the axis itself. This section examines how to control these properties of a graph.

Axis Labels and Titles

Arguably, the axis label is the most important property of an axis, which needs to be set in nearly all cases. We can set the axis labels using the set_xlabel and set_ylabel methods: they both take a string with the label as the first argument. In addition, the optional labelpad argument specifies the spacing from the axis to the label in units of points. This padding is occasionally necessary to avoid overlap between the axis label and the axis tick labels. The set_xlabel and set_ylabel methods also take additional arguments for setting text properties, such as color, fontsize, and fontname. The following code, which produces Figure 4-12, demonstrates how to use the set_xlabel and set_ylabel methods and the keyword arguments discussed here.

```
In [14]: x = np.linspace(0, 50, 500)
    ...: y = np.sin(x) * np.exp(-x/10)
    ...:
    ...: fig, ax = plt.subplots(figsize=(8, 2),
    ...:                        subplot_kw={'facecolor': "#ebf5ff"})
    ...:
    ...: ax.plot(x, y, lw=2)
    ...:
    ...: ax.set_xlabel("x", labelpad=5, fontsize=18, fontname='serif',
    ...:               color="blue")
    ...: ax.set_ylabel("f(x)", labelpad=15, fontsize=18, fontname='serif',
    ...:               color="blue")
    ...: ax.set_title("axis labels and title example", fontsize=16,
    ...:              fontname='serif', color="blue")
```

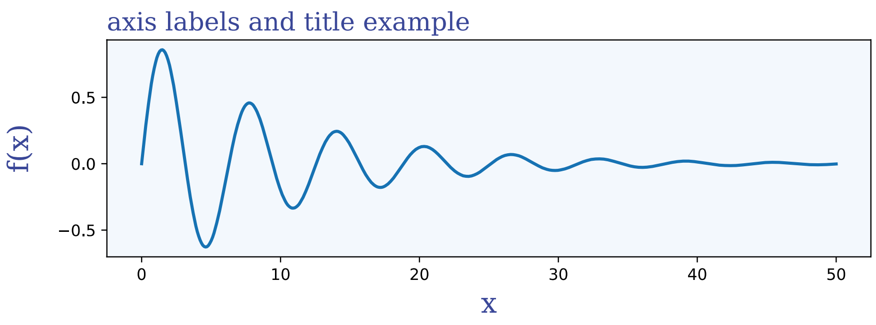


Figure 4-12. *Graph demonstrating the result of using set_xlabel and set_ylabel for setting the x and y axes labels*

In addition to labels on the x and y axes, we can also set the title of an Axes object using the set_title method. This method takes mostly the same arguments as set_xlabel and set_ylabel, except the loc argument, which can be assigned to 'left', 'centered', to 'right', and which dictates that the title is to be left aligned, centered, or right aligned.

Axis Range

By default, the range of the *x* and *y* axes of a Matplotlib is automatically adjusted to the data plotted in the Axes object. In many cases, these default ranges are sufficient, but in some situations, explicitly setting the axis ranges may be necessary. In such cases, we can use the set_xlim and set_ylim methods of the Axes object. These methods take two arguments that specify the lower and upper limit to be displayed on the axis, respectively. An alternative to `set_xlim` and `set_ylim` is the `axis` method, which, for example, accepts the string argument `'tight'`, for a coordinate range that tightly fits the lines it contains and `'equal'` for a coordinate range where one unit length along each axis corresponds to the same number of pixels (i.e., a ratio preserving coordinate system).

It is also possible to use the `autoscale` method to selectively turn on and off autoscaling by passing `True` and `False` as the first argument for the *x* and/or *y-axis* by setting its `axis` argument to `'x'`, `'y'`, or `'both'`. The following example shows how to use these methods to control axis ranges. The resulting graphs are shown in Figure 4-13.

```
In [15]: x = np.linspace(0, 30, 500)
    ...: y = np.sin(x) * np.exp(-x/10)
    ...:
    ...:
    ...: fig, axes = plt.subplots(1, 3, figsize=(9, 3),
    ...:                          subplot_kw={'facecolor': "#ebf5ff"})
    ...:
    ...: axes[0].plot(x, y, lw=2)
    ...: axes[0].set_xlim(-5, 35)
    ...: axes[0].set_ylim(-1, 1)
    ...: axes[0].set_title("set_xlim / set_y_lim")
    ...:
    ...: axes[1].plot(x, y, lw=2)
    ...: axes[1].axis('tight')
    ...: axes[1].set_title("axis('tight')")
    ...:
    ...: axes[2].plot(x, y, lw=2)
    ...: axes[2].axis('equal')
    ...: axes[2].set_title("axis('equal')")
```

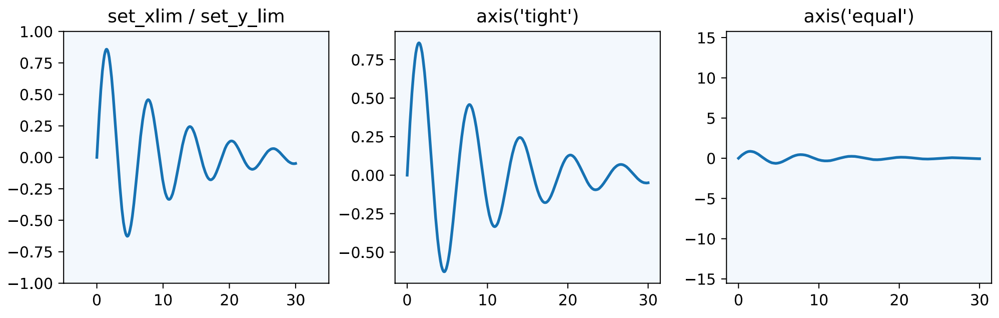


Figure 4-13. *Graphs that show the result of using the `set_xlim`, `set_ ylim`, and `axis` methods for setting the axis ranges that are shown in a graph*

Axis Ticks, Tick Labels, and Grids

The final basic properties of the axis that remain to be specified are the placement of axis ticks and the placement and formatting of the corresponding tick labels. The axis ticks are an important part of the overall appearance of a graph. When preparing publication and production-quality graphs, detailed control over the axis ticks is often necessary. Matplotlib module `mpl.ticker` provides a general and extensible tick management system that gives full control of the tick placement. Matplotlib distinguishes between major ticks and minor ticks. By default, every major tick has a corresponding label, and the distances between major ticks may be further marked with minor ticks that do not have labels, although this feature must be explicitly turned on. Figure 4-14 illustrates major and minor ticks.

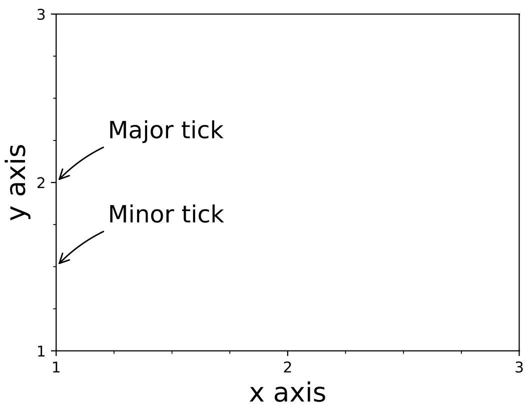


Figure 4-14. *The difference between major and minor ticks*

When configuring graph ticks, a common design requirement is determining where the major ticks with labels should be placed along the coordinate axis. The `mpl.ticker` module provides classes for different tick placement strategies. For example, the `mpl.ticker.MaxNLocator` can set the maximum number of ticks (at unspecified locations), the `mpl.ticker.MultipleLocator` can be used for setting ticks at multiples of a given base, and the `mpl.ticker.FixedLocator` can be used to place ticks at explicitly specified coordinates. To change the ticker strategy, use the `set_major_locator` and the `set_minor_locator` methods in `Axes.xaxis` and `Axes.yaxis`. These methods accept an instance of a ticker class defined in `mpl.ticker` or a custom class that is derived from one of those classes.

When explicitly specifying tick locations, we can also use the `set_xticks` and `set_yticks` methods, which accept a list of coordinates for where to place major ticks. In this case, it is also possible to set custom labels for each tick using the `set_xticklabels` and `set_yticklabels` methods, which expect lists of strings to be used as labels for the corresponding ticks. It is a good idea to use generic tick placement strategies, for example, `mpl.ticker.MaxNLocator`, because they dynamically adjust if the coordinate range is changed, whereas explicit tick placement using `set_xticks` and `set_yticks` then would require manual code updates. However, when the exact placement of ticks must be controlled, then `set_xticks` and `set_yticks` are convenient methods.

The following code demonstrates how to change the default tick placement using combinations of the methods discussed in this section, and the resulting graphs are shown in Figure 4-15.

```
In [16]: x = np.linspace(-2 * np.pi, 2 * np.pi, 500)
    ...: y = np.sin(x) * np.exp(-x**2/20)
    ...:
    ...: fig, axes = plt.subplots(1, 4, figsize=(12, 3))
    ...:
    ...: axes[0].plot(x, y, lw=2)
    ...: axes[0].set_title("default ticks")
    ...: axes[1].plot(x, y, lw=2)
    ...: axes[1].set_title("set_xticks")
    ...: axes[1].set_yticks([-1, 0, 1])
    ...: axes[1].set_xticks([-5, 0, 5])
    ...:
    ...: axes[2].plot(x, y, lw=2)
    ...: axes[2].set_title("set_major_locator")
    ...: axes[2].xaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
    ...: axes[2].xaxis.set_minor_locator(mpl.ticker.MaxNLocator(8))
    ...: axes[2].yaxis.set_major_locator(mpl.ticker.FixedLocator([-1, 0, 1]))
    ...: axes[2].yaxis.set_minor_locator(mpl.ticker.MaxNLocator(8))
    ...:
    ...: axes[3].plot(x, y, lw=2)
    ...: axes[3].set_title("set_xticklabels")
    ...: axes[3].set_yticks([-1, 0, 1])
    ...: axes[3].set_xticks([-2 * np.pi, -np.pi, 0, np.pi, 2 * np.pi])
    ...: axes[3].set_xticklabels([r'$-2\pi$', r'$-\pi$', 0, r'$\pi$', r'$2\pi$'])
    ...: x_minor_ticker = mpl.ticker.FixedLocator(
    ...:     [-3 * np.pi / 2, -np.pi / 2, 0, np.pi / 2, 3 * np.pi / 2])
    ...: axes[3].xaxis.set_minor_locator(x_minor_ticker)
    ...: axes[3].yaxis.set_minor_locator(mpl.ticker.MaxNLocator(4))
```

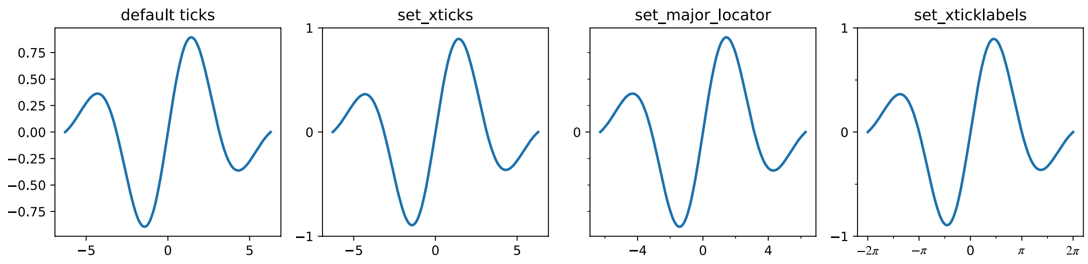


***Figure 4-15.** Graphs that demonstrate different ways of configuring the placement and appearance of major and minor ticks along the x axis and the y axis*

A commonly used design element in graphs is grid lines, which are intended as a visual guide when reading values from the graph. Grids and grid lines are closely related to axis ticks since they are drawn at the same coordinate values and are essentially extensions of the ticks that span across the graph. In Matplotlib, we can turn on axis grids using the `grid` method of an axes object. The grid method takes optional keyword arguments used to control the grid's appearance. For example, like many of the plot functions in Matplotlib, the `grid` method accepts the `color`, `linestyle`, and `linewidth` arguments for specifying the properties of the grid lines. In addition, it takes the `which` and `axis` arguments, which can be assigned values `'major'`, `'minor'`, or `'both'`, and `'x'`, `'y'`, or `'both'`, respectively. These arguments indicate which ticks the given style is to be applied along an axis. If several different styles for the grid lines are required, multiple calls to `grid` can be used, with different values of `which` and `axis`. For an example of how to add grid lines and style them in different ways, see the following code listing, which produces the graphs shown in Figure 4-16.

```
In [17]: fig, axes = plt.subplots(1, 3, figsize=(12, 4))
    ...: x_major_ticker = mpl.ticker.MultipleLocator(4)
    ...: x_minor_ticker = mpl.ticker.MultipleLocator(1)
    ...: y_major_ticker = mpl.ticker.MultipleLocator(0.5)
    ...: y_minor_ticker = mpl.ticker.MultipleLocator(0.25)
    ...:
    ...: for ax in axes:
    ...:     ax.plot(x, y, lw=2)
    ...:     ax.xaxis.set_major_locator(x_major_ticker)
    ...:     ax.yaxis.set_major_locator(y_major_ticker)
    ...:     ax.xaxis.set_minor_locator(x_minor_ticker)
    ...:     ax.yaxis.set_minor_locator(y_minor_ticker)
    ...:
    ...: axes[0].set_title("default grid")
    ...: axes[0].grid()
    ...:
    ...: axes[1].set_title("major/minor grid")
    ...: axes[1].grid(color="blue", which="both", linestyle=':', linewidth=0.5)
    ...:
    ...: axes[2].set_title("individual x/y major/minor grid")
    ...: axes[2].grid(color="grey", which="major", axis='x',
    ...:              linestyle='-', linewidth=0.5)
    ...: axes[2].grid(color="grey", which="minor", axis='x',
    ...:              linestyle=':', linewidth=0.25)
    ...: axes[2].grid(color="grey", which="major", axis='y',
    ...:              linestyle='-', linewidth=0.5)
```

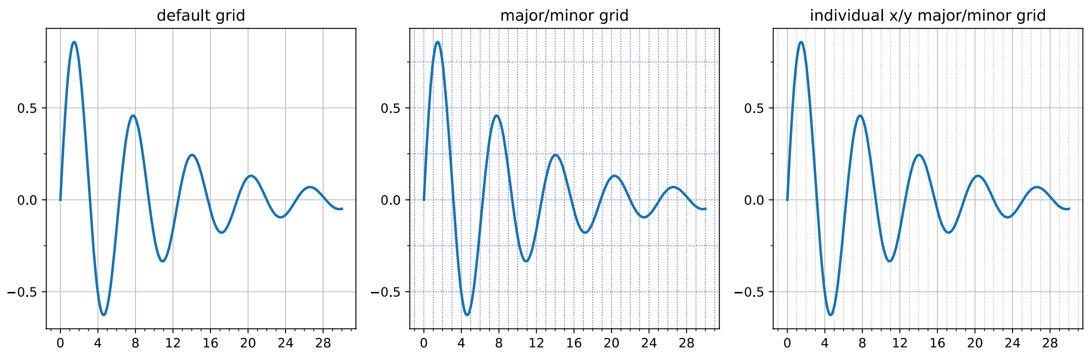


Figure 4-16. *Graphs demonstrating the result of using grid lines*

In addition to controlling the tick placements, the Matplotlib `mpl.ticker` module also provides classes for customizing the tick labels. For example, the `ScalarFormatter` from the `mpl.ticker` module can be used to set several useful properties related to displaying tick labels with scientific notation, for displaying axis labels for large numerical values. If scientific notation is activated using the `set_scientific` method, we can control the threshold for when scientific notation is used with the `set_powerlimits` method (by default, tick labels for small numbers are not displayed using the scientific notation). We can use the `useMathText=True` argument when creating the `ScalarFormatter` instance to have the exponents shown in math style rather than code style exponents (e.g., `1e10`). The `formatter` object is applied to an Axes object using the `set_major_formatter` method. See the following code for an example of using scientific notation in tick labels. The resulting graphs are shown in Figure 4-17.

```
In [19]: fig, axes = plt.subplots(1, 2, figsize=(8, 3))
    ...:
    ...: x = np.linspace(0, 1e5, 100)
    ...: y = x ** 2
    ...:
    ...: axes[0].plot(x, y, 'b.')
    ...: axes[0].set_title("default labels", loc='right')
    ...:
```

```
    ...: axes[1].plot(x, y, 'b')
    ...: axes[1].set_title("scientific notation labels", loc='right')
    ...:
    ...: formatter = mpl.ticker.ScalarFormatter(useMathText=True)
    ...: formatter.set_scientific(True)
    ...: formatter.set_powerlimits((-1,1))
    ...: axes[1].xaxis.set_major_formatter(formatter)
    ...: axes[1].yaxis.set_major_formatter(formatter)
```

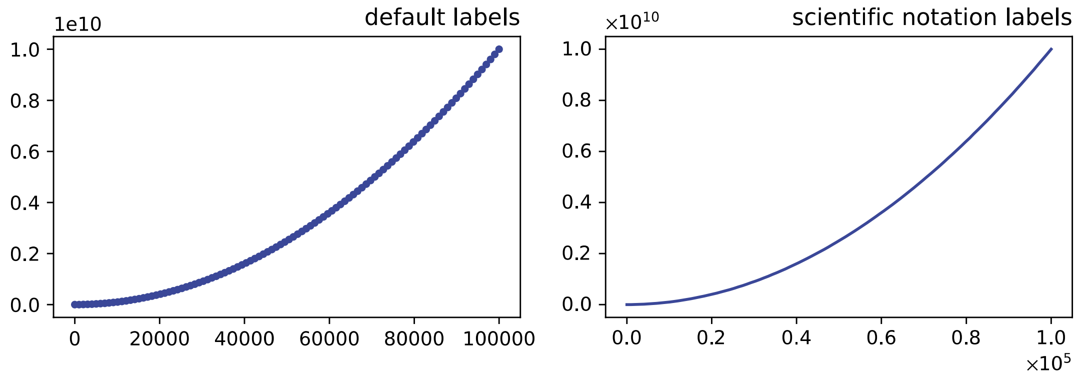


Figure 4-17. *Graphs with tick labels in scientific notation. The left panel uses the default label formatting, while the right panel uses tick labels in scientific notation, rendered as math text*

Log Plots

It is useful to work with logarithmic coordinate systems in visualizations of data that spans several orders of magnitude. In Matplotlib, there are several plot functions for graphing functions in such coordinate systems, for example, `loglog`, `semilogx`, and `semilogy`, which use logarithmic scales for both the *x* and *y* axes, for only the *x axis*, and only the *y axis*, respectively. Apart from the logarithmic axis scales, these functions behave similarly to the standard plot method. An alternative approach is using the standard plot method and separately configuring the axis scales to be logarithmic using the `set_xscale` and/or `set_yscale` method with `'log'` as the first argument. Examples of using methods to produce log-scale plots are shown in the following code listing, and the resulting graphs are shown in Figure 4-18.

```
In [20]: fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...:
    ...: x = np.linspace(0, 1e3, 100)
    ...: y1, y2 = x**3, x**4
    ...:
    ...: axes[0].set_title('loglog')
    ...: axes[0].loglog(x, y1, 'b', x, y2, 'r')
    ...:
    ...: axes[1].set_title('semilogy')
    ...: axes[1].semilogy(x, y1, 'b', x, y2, 'r')
    ...:
    ...: axes[2].set_title('plot / set_xscale / set_yscale')
    ...: axes[2].plot(x, y1, 'b', x, y2, 'r')
    ...: axes[2].set_xscale('log')
    ...: axes[2].set_yscale('log')
```

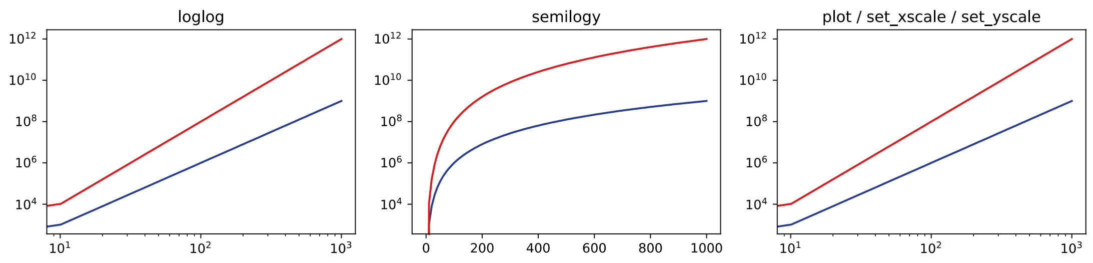


Figure 4-18. *Examples of log-scale plots*

Twin Axes

An interesting trick with axes that Matplotlib provides is the twin axis feature, which allows displaying two independent axes overlaid on each other. This is useful when plotting two different quantities, for example, with different units, within the same graph. A simple example demonstrating this feature is shown as follows, and the resulting graph is shown in Figure 4-19. Let's use the `twinx` method (there is also a `twiny` method) to produce a second `Axes` instance with a shared *x axis* and a new independent *y axis*, which is displayed on the right side of the graph.

```
In [21]: fig, ax1 = plt.subplots(figsize=(8, 4))
    ...:
    ...: r = np.linspace(0, 5, 100)
    ...: a = 4 * np.pi * r ** 2  # area
    ...: v = (4 * np.pi / 3) * r ** 3  # volume
    ...:
    ...: ax1.set_title("surface area and volume of a sphere", fontsize=16)
    ...: ax1.set_xlabel("radius [m]", fontsize=16)
    ...:
    ...: ax1.plot(r, a, lw=2, color="blue")
    ...: ax1.set_ylabel(r"surface area ($m^2$)", fontsize=16, color="blue")
    ...: for label in ax1.get_yticklabels():
    ...:     label.set_color("blue")
    ...:
    ...: ax2 = ax1.twinx()
    ...: ax2.plot(r, v, lw=2, color="red")
    ...: ax2.set_ylabel(r"volume ($m^3$)", fontsize=16, color="red")
    ...: for label in ax2.get_yticklabels():
    ...:    label.set_color("red")
```

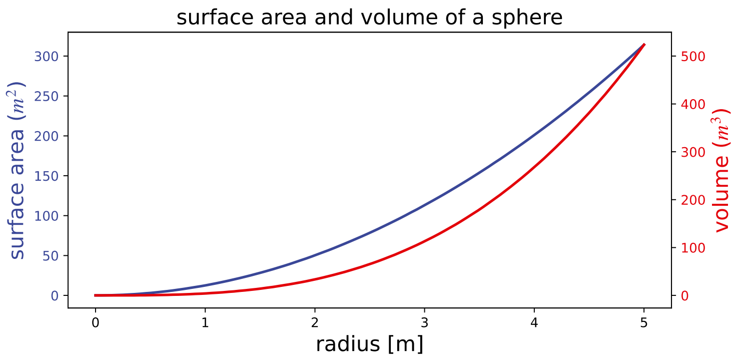


Figure 4-19. *Example of a graph with twin axes*

Spines

In all graphs generated so far, there was always a box surrounding the Axes region. This is indeed a common style for scientific and technical graphs, but in some cases, for example, moving these coordinate lines may be desired when representing schematic graphs. The lines that make up the surrounding box are called axis spines in Matplotlib, and we can use the Axes.spines attribute to change their properties. For example, we might want to remove the top and the right spines and move the spines to coincide with the origin of the coordinate systems.

The spines attribute of the Axes object is a dictionary with the keys right, left, top, and bottom that can be used to access each spine individually. We can use the set_color method to set the color to 'None' to indicate that a particular spine should not be displayed. In this case, we also need to remove the ticks associated with that spine, using the set_ticks_position method of Axes.xaxis and Axes.yaxis (which accepts the 'both', 'top', or 'bottom' and 'both', 'left', or 'right' arguments, respectively). These methods allow we to transform the surrounding box to *x* and *y* coordinate axes, as demonstrated in the following example. The resulting graph is shown in Figure 4-20.

```
In [22]: x = np.linspace(-10, 10, 500)
    ...: y = np.sin(x) / x
    ...:
    ...: fig, ax = plt.subplots(figsize=(8, 4))
    ...:
    ...: ax.plot(x, y, linewidth=2)
    ...:
    ...: # remove top and right spines
    ...: ax.spines['right'].set_color('none')
    ...: ax.spines['top'].set_color('none')
    ...:
    ...: # remove top and right spine ticks
    ...: ax.xaxis.set_ticks_position('bottom')
    ...: ax.yaxis.set_ticks_position('left')
    ...:
    ...: # move bottom and left spine to x = 0 and y = 0
    ...: ax.spines['bottom'].set_position(('data', 0))
    ...: ax.spines['left'].set_position(('data', 0))
    ...:
```

```
   ...: ax.set_xticks([-10, -5, 5, 10])
   ...: ax.set_yticks([0.5, 1])
   ...:
   ...: # give each label a solid background of white,
   ...: # to not overlap with the plot line
   ...: for label in ax.get_xticklabels() + ax.get_yticklabels():
   ...:     label.set_bbox({'facecolor': 'white',
   ...:                     'edgecolor': 'white'})
```

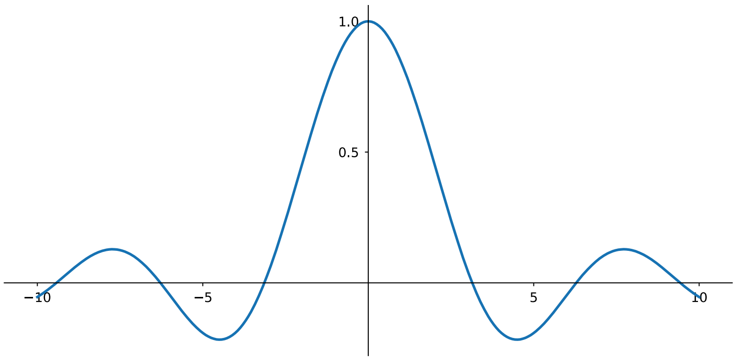


Figure 4-20. *Example of a graph with axis spines*

Advanced Axes Layouts

So far, plt.figure, Figure.make_axes, and plt.subplots have been used to create new Figure and Axes instances, which we then used for producing graphs. In scientific and technical visualization, it is common to pack together multiple figures in different panels, for example, in a grid layout. Matplotlib has functions for automatically creating Axes objects and placing them on a figure canvas using various layout strategies. We have already used the plt.subplots function to generate a uniform grid of Axes objects. This section explores additional features of the plt.subplots function and introduces the subplot2grid and GridSpec layout managers, which are more flexible in how the Axes objects are distributed within a figure canvas.

Insets

Before diving into using more advanced Axes layout managers, it is worth taking a step back and considering an important use for the first approach used to add Axes instances to a figure canvas: the Figure.add_axes method. This approach is well suited for creating an *inset*, a smaller graph displayed within the region of another graph. Insets are, for example, frequently used for displaying a magnified region of special interest in the larger graph or for displaying some related graphs of secondary importance.

In Matplotlib, we can place additional Axes objects at arbitrary locations within a figure canvas, even if they overlap with existing Axes objects. To create an inset, add a new Axes object with Figure.make_axes and the (figure canvas) coordinates for where the inset should be placed. The following code produces a typical example of a graph with an inset, as shown in Figure 4-21. When creating the Axes object for the inset, it may be useful to use the facecolor='none' argument, which indicates that there should be no background color and that the Axes background of the inset should be transparent.

```
In [23]: fig = plt.figure(figsize=(8, 4))
    ...:
    ...: def f(x):
    ...:     return 1/(1 + x**2) + 0.1/(1 + ((3 - x)/0.1)**2)
    ...:
    ...: def plot_and_format_axes(ax, x, f, fontsize):
    ...:     ax.plot(x, f(x), linewidth=2)
    ...:     ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(5))
    ...:     ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
    ...:     ax.set_xlabel(r"$x$", fontsize=fontsize)
    ...:     ax.set_ylabel(r"$f(x)$", fontsize=fontsize)
    ...:
    ...: # main graph
    ...: ax = fig.add_axes([0.1, 0.15, 0.8, 0.8], facecolor="#f5f5f5")
    ...: x = np.linspace(-4, 14, 1000)
    ...: plot_and_format_axes(ax, x, f, 18)
    ...:
    ...: x0, x1 = 2.5, 3.5
    ...: ax.axvline(x0, ymax=0.3, color="grey", linestyle=":")
    ...: ax.axvline(x1, ymax=0.3, color="grey", linestyle=":")
    ...:
    ...: # inset
    ...: ax_insert = fig.add_axes([0.5, 0.5, 0.38, 0.42], facecolor='none')
    ...: x = np.linspace(x0, x1, 1000)
    ...: plot_and_format_axes(ax_insert, x, f, 14)
```

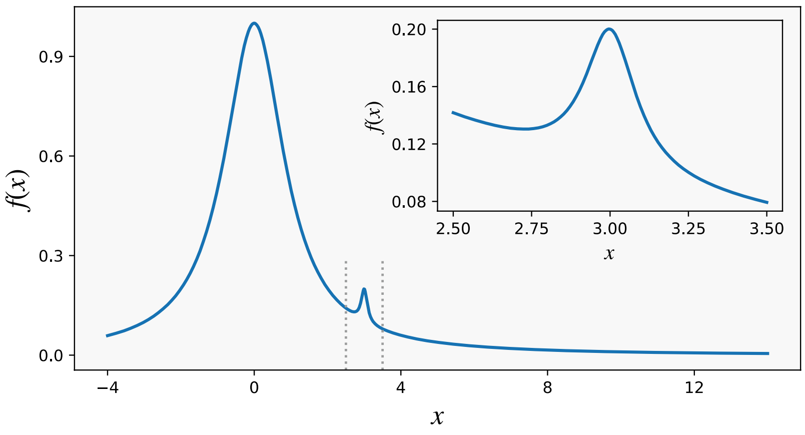


Figure 4-21. *Example of a graph with an inset*

Subplots

We have already used `plt.subplots` extensively. It returns a tuple with a `Figure` instance and a NumPy array with the Axes objects for each row and column requested in the function call. It is often the case when plotting grids of subplots that either the *x* or the *y* axis, or both, is shared among the subplots. Using the `sharex` and `sharey` arguments to `plt.subplots` can be useful in such situations since it prevents the same axis labels from being repeated across multiple Axes.

It is also worth noting that the dimension of the NumPy array with Axes instances returned by `plt.subplots` is “squeezed” by default; the dimensions with length 1 are removed from the array. If both the requested numbers of column and row are greater than one, then a two-dimensional array is returned, but if either (or both) the number of columns or rows is one, then a one-dimensional (or scalar, i.e., the only Axes object itself) is returned. We can turn off the squeezing of the dimensions of the NumPy arrays by passing the `squeeze=False` argument to the `plt.subplots` function. In this case, the axes variable in `fig, axes = plt.subplots(nrows, ncols)` is always a two-dimensional array.

A final touch of configurability can be achieved using the `plt.subplots_adjust` function, which allows explicitly setting the left, right, bottom, and top coordinates of the overall Axes grid, as well as the width (`wspace`) and height spacing (`hspace`) between Axes instances in the grid. The following code and the corresponding Figure 4-22 provide a step-by-step example of setting up an Axes grid with shared x and y axes and adjusted Axes spacing.

```
In [24]: fig, axes = plt.subplots(2, 2, figsize=(6, 6),
    ...:                          sharex=True, sharey=True, squeeze=False)
    ...:
    ...: x1 = np.random.randn(100)
    ...: x2 = np.random.randn(100)
    ...:
    ...: axes[0, 0].set_title("Uncorrelated")
    ...: axes[0, 0].scatter(x1, x2)
    ...:
    ...: axes[0, 1].set_title("Weakly positively correlated")
    ...: axes[0, 1].scatter(x1, x1 + x2)
    ...:
    ...: axes[1, 0].set_title("Weakly negatively correlated")
    ...: axes[1, 0].scatter(x1, -x1 + x2)
    ...:
    ...: axes[1, 1].set_title("Strongly correlated")
    ...: axes[1, 1].scatter(x1, x1 + 0.15 * x2)
    ...:
    ...: axes[1, 1].set_xlabel("x")
    ...: axes[1, 0].set_xlabel("x")
    ...: axes[0, 0].set_ylabel("y")
    ...: axes[1, 0].set_ylabel("y")
    ...:
    ...: plt.subplots_adjust(left=0.1, right=0.95, bottom=0.1, top=0.95,
    ...:                     wspace=0.1, hspace=0.2)
```

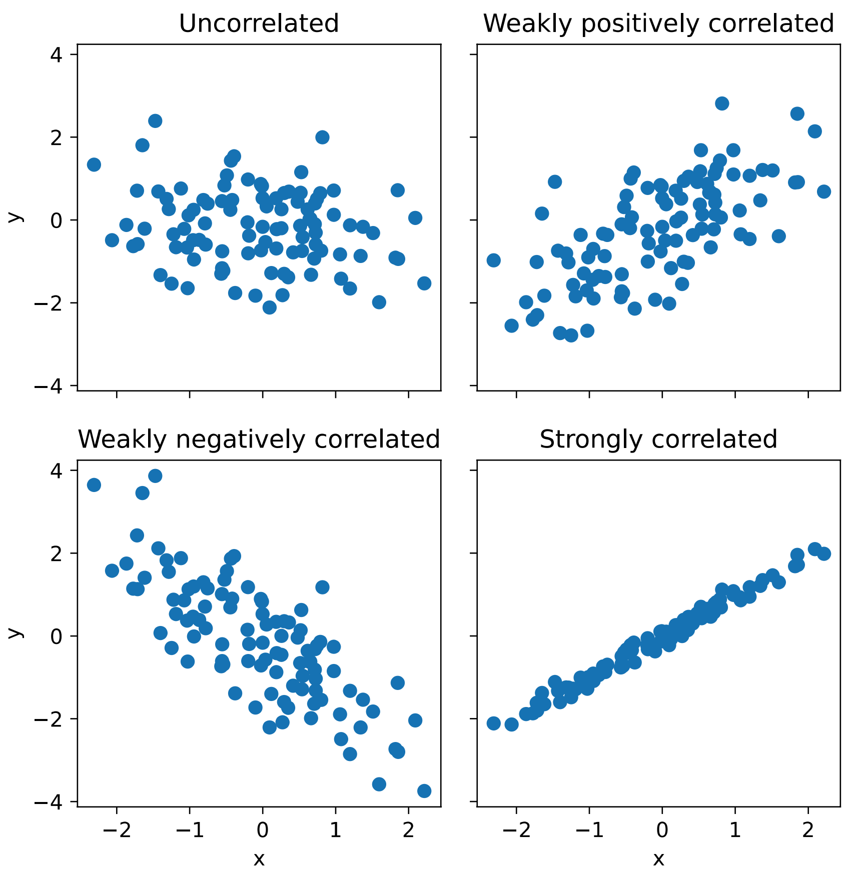


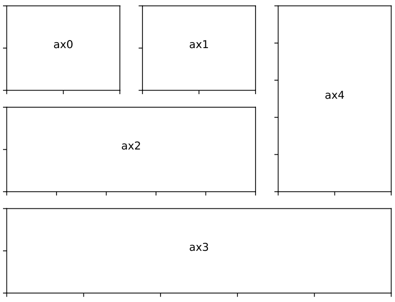
***Figure 4-22.** Example graph using `plt.subplot` and `plt.subplot_adjust`*

Subplot2grid

The `plt.subplot2grid` function is an intermediary between `plt.subplots` and `gridspec` (see the next section) that provides a more flexible Axes layout management than `plt.subplots` while at the same time being simpler to use than `gridspec`. In particular, `plt.subplot2grid` can create grids with Axes instances that span multiple rows and/or columns. The `plt.subplot2grid` takes two mandatory arguments: the first argument is the shape of the Axes grid in the form on a tuple `(nrows, ncols)`, and the second argument is a tuple `(row, col)` that specifies the starting position within the grid. The two optional keyword arguments `colspan` and `rowspan` can indicate how many rows and columns the new Axes instance should span. An example of how to use the `plt.subplot2grid` function is given in Table 4-3. Note that each call to the `plt.subplot2grid` function results in one new Axes instance, unlike `plt.subplots`, which creates all Axes instances in one function call and returns them in a NumPy array.

***Table 4-3.** Example of a Grid Layout Created with `plt.subplot2grid` and the Corresponding Code*

Axes Grid Layout	Code

	`ax0 = plt.subplot2grid((3, 3), (0, 0))`
	`ax1 = plt.subplot2grid((3, 3), (0, 1))`
	`ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)`
	`ax3 = plt.subplot2grid((3, 3), (2, 0), colspan=3)`
	`ax4 = plt.subplot2grid((3, 3), (0, 2), rowspan=2)`

GridSpec

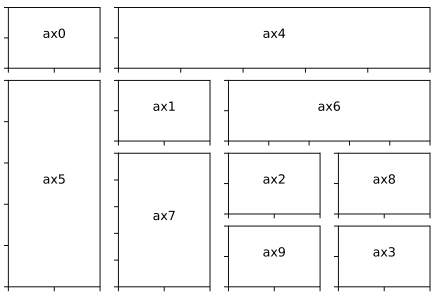
The final grid layout manager covered here is `GridSpec` from the `mpl.gridspec` module. This is the most general grid layout manager in Matplotlib, and it allows the creation of grids where not all rows and columns have equal width and height, which is not easily achieved with the grid layout managers used earlier in this chapter.

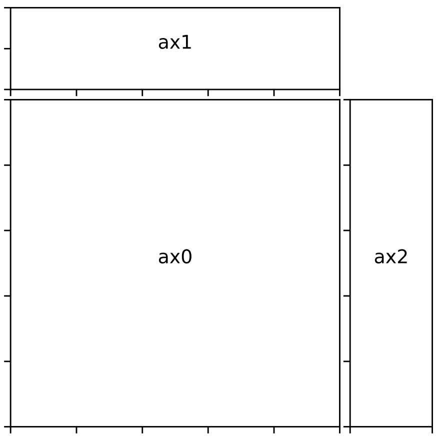
A `GridSpec` object is only used to specify the grid layout and does not create any `Axes` objects. When creating a new instance of the `GridSpec` class, we must specify the number of rows and columns in the grid. Like for other grid layout managers, we can also set the position of the grid using the keyword arguments `left`, `bottom`, `right`, and `top`, and we can set the width and height spacing between subplots using `wspace` and `hspace`. Additionally, `GricSpec` allows specifying the relative width and heights of columns and rows using the `width_ratios` and `height_ratios` arguments. These should both be lists with relative weights for each column and row size in the grid. For example, to generate a grid with two rows and two columns, where the first row and column is twice as big as the second row and column, we could use `mpl.gridspec.GridSpec(2, 2, width_ratios=[2, 1], height_ratios=[2, 1])`.

Once a GridSpec instance has been created, we can use the `Figure.add_subplot` method to create `Axes` objects and place them on a figure canvas. As an argument to `add_subplot`, we need to pass an `mpl.gridspec.SubplotSpec` instance, which we can generate from the `GridSpec` object using an array-like indexing: for example, given a `GridSpec` instance `gs`, we obtain a `SubplotSpec` instance for the upper-left grid element using `gs[0, 0]` and for a `SubplotSpec` instance that covers the first row we use `gs[:, 0]` and so on. Table 4-4 provides concrete examples of how to use `GridSpec` and `add_subplot` to create an `Axes` instance.

Table 4-4. *Examples of How to Use the Subplot Grid Manager `mpl.gridspec.GridSpec`*

Axes Grid Layout	Code

 ax0
ax4
ax1
ax6
ax5
ax2
ax8
ax7
ax9
ax3
	`fig = plt.figure(figsize=(6, 4))`
	`gs = mpl.gridspec.GridSpec(4, 4)`
	`ax0 = fig.add_subplot(gs[0, 0])`
	`ax1 = fig.add_subplot(gs[1, 1])`
	`ax2 = fig.add_subplot(gs[2, 2])`
	`ax3 = fig.add_subplot(gs[3, 3])`
	`ax4 = fig.add_subplot(gs[0, 1:])`
	`ax5 = fig.add_subplot(gs[1:, 0])`
	`ax6 = fig.add_subplot(gs[1, 2:])`
	`ax7 = fig.add_subplot(gs[2:, 1])`
	`ax8 = fig.add_subplot(gs[2, 3])`
	`ax9 = fig.add_subplot(gs[3, 2])`

	`fig = plt.figure(figsize=(4, 4))`
	`gs = mpl.gridspec.GridSpec(`
	`      2, 2,`
	`      width_ratios=[4, 1],`
	`      height_ratios=[1, 4],`
	`      wspace=0.05, hspace=0.05)`
	`ax0 = fig.add_subplot(gs[1, 0])`
	`ax1 = fig.add_subplot(gs[0, 0])`
	`ax2 = fig.add_subplot(gs[1, 1])`

Colormap Plots

We have so far only worked with graphs of univariate functions or, equivalently, twodimensional data in *x*-*y* format. The two-dimensional Axes objects that are used for this purpose can also be used to visualize bivariate functions or three-dimensional data on *x*-*y*-z format, using *color maps* (or *heat maps*), where each pixel in the Axes area is colored according to the *z* value corresponding to that point in the coordinate system. Matplotlib provides the `pcolor` and `imshow` functions for these types of plots, and the `contour` and `contourf` functions graph data in the same format by drawing contour lines rather than color maps. Examples of graphs generated with these functions are shown in Figure 4-23.

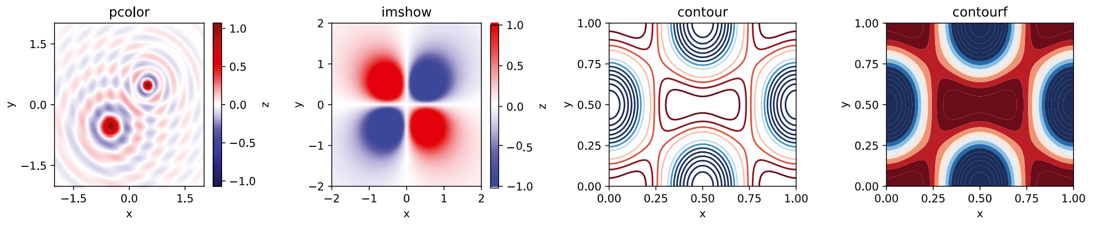


Figure 4-23. *Example graphs generated with `pcolor`, `imshow`, `contour`, and `contourf`*

To produce a colormap graph, for example, using `pcolor`, we must prepare the data in the appropriate format. While standard two-dimensional graphs expect one-dimensional coordinate arrays with *x* and *y* values, in the present case, we need to use two-dimensional coordinate arrays, as, for example, generated using the NumPy `meshgrid` function. To plot a bivariate function or data with two dependent variables, we start by defining one-dimensional coordinate arrays, `x` and `y`, that span the desired coordinate range or correspond to the values for which data is available. The `x` and `y` arrays can then be passed to the `np.meshgrid` function, which produces the required two-dimensional coordinate arrays `X` and `Y`. If necessary, we can use NumPy array computations with `X` and `Y` to evaluate bivariate functions to obtain a data array `Z`, as done in lines 1 to 3 in cell `In [25]` (see the following).

Once the two-dimensional coordinate and data arrays are prepared, they are easily visualized using, for example, `pcolor`, `contour`, or `contourf`, by passing the `X`, `Y`, and `Z` arrays as the first three arguments. The `imshow` method works similarly but only expects the data array `Z` as an argument, and the relevant coordinate ranges must instead be set using the `extent` argument, which should be set to a list in the format `[xmin, xmax, ymin, ymax]`. Additional keyword arguments important for controlling the appearance of colormap graphs are `vmin`, `vmax`, `norm`, and `cmap`: the `vmin` and `vmax` can be used to set the range of values mapped to the color axis. This can also be achieved by setting `norm=mpl.colors.Normalize(vmin, vmax)`. The `cmap` argument specifies a color map for mapping the data values to colors in the graph. This argument can either be a string with a predefined colormap name or a colormap instance. The predefined color maps in Matplotlib are available in `mpl.cm`. Try `help(mpl.cm)` or to autocomplete in IPython on the `mpl.cm` module for a full list of available color maps.[5]

The last piece required for a complete colormap plot is the colorbar element, which allows the viewer to read off the numerical values that different colors correspond to. In Matplotlib we can use the `plt.colorbar` function to attach a colorbar to a plotted colormap graph. It takes a handle to the plot as first argument, and it takes two optional arguments `ax` and `cax`, which can be used to control where in the graph the colorbar is to appear. If `ax` is given, the space is taken from this `Axes` object for the new colorbar. If, on the other hand, `cax` is given, then the colorbar draws on this `Axes` object. A colorbar instance `cb` has its own axis object, and the standard methods for setting axis attributes can be used on the `cb.ax` object, and we can use, for example, the `set_label`, `set_ticks`, and `set_ticklabels` method in the same manner as for *x* and *y* axes.

The steps outlined in the previous paragraphs are shown in the following code, and the resulting graph is shown in Figure 4-24. The `imshow`, `contour`, and `contourf` functions can be used in a nearly similar manner, although these functions take additional arguments for controlling their characteristic properties. For example, the `contour` and `contourf` functions additionally take an argument `N` that specifies the number of contour lines to draw.

```
In [25]: x = y = np.linspace(-10, 10, 150)
    ...: X, Y = np.meshgrid(x, y)
    ...: Z = np.cos(X) * np.cos(Y) * np.exp(-(X/5)**2-(Y/5)**2)
    ...:
    ...: fig, ax = plt.subplots(figsize=(6, 5))
    ...:
    ...: norm = mpl.colors.Normalize(-abs(Z).max(), abs(Z).max())
    ...: p = ax.pcolor(X, Y, Z, norm=norm, cmap=mpl.cm.bwr)
    ...:
    ...: ax.axis('tight')
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$y$", fontsize=18)
    ...: ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
    ...: ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(4))
```

[5] A nice visualization of all the available color maps is available at `http://scipy-cookbook.readthedocs.io/items/Matplotlib_Show_colormaps.html`. This page also describes how to create new color maps.

```
    ...:
    ...: cb = fig.colorbar(p, ax=ax)
    ...: cb.set_label(r"$z$", fontsize=18)
    ...: cb.set_ticks([-1, -.5, 0, .5, 1])
```

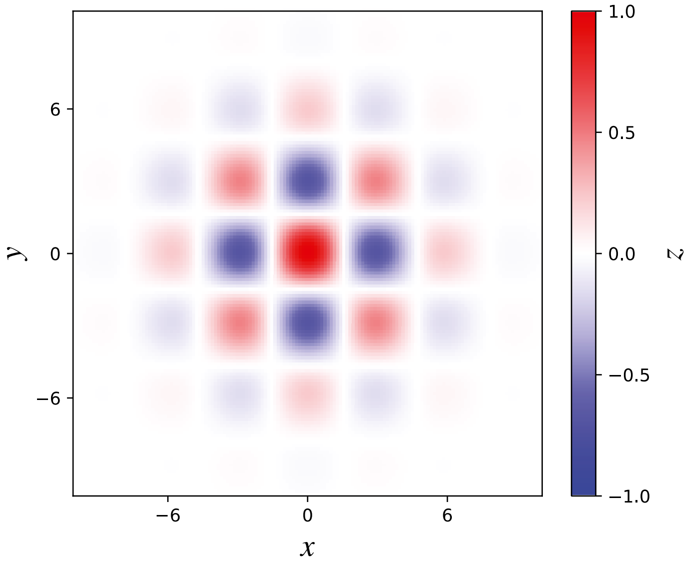


***Figure 4-24.** Example using `pcolor` to produce a colormap graph*

3D Plots

The colormap graphs discussed in the previous section were used to visualize data with two dependent variables by color-coding data in 2D graphs. Another way of visualizing the same type of data is to use 3D graphs, where a third axis, *z,* is introduced, and the graph is displayed in a perspective on the screen. In Matplotlib, drawing 3D graphs requires using a different axes object, namely, the `Axes3D` object available from the `mpl_toolkits.mplot3d` module. We can create a 3D-aware `Axes` instance explicitly using the constructor of the `Axes3D` class, by passing a `Figure` instance as an argument: `ax = Axes3D(fig)`. Alternatively, we can use the `add_subplot` function with the `projection='3d'` argument.

```
ax = fig.add_subplot(1, 1, 1, projection='3d')
```

Or use `plt.subplots` with the `subplot_kw={'projection': '3d'}` argument.

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6), subplot_kw={'projection': '3d'})
```

In this way, we can use all of the axes layout approaches previously used for 2D graphs, if only we specify the `projection` argument appropriately. Note that using `add_subplot`, it is possible to mix axes objects with 2D and 3D projections within the same figure, but when using `plt.subplots`, the `subplot_kw` argument applies to all the subplots added to a figure.

Having created and added 3D-aware `Axes` instances to a figure, for example, using one of the methods described in the previous paragraph, the `Axes3D` class methods—such as `plot_surface`, `plot_wireframe`, and `contour`— can be used to plot data as surfaces in a 3D perspective. These functions are used in a manner that is nearly the same as how the color map was used in the previous section: these 3D plotting functions all take two-dimensional coordinates and data arrays X, Y, and Z as first arguments. Each function also takes additional parameters for tuning specific properties. For example, the plot_surface function takes the rstride and cstride arguments (row and column stride) for selecting data from the input arrays (to avoid data points that are too dense). The contour and contourf functions take optional arguments zdir and offset, which are used to select a projection direction (the allowed values are "x," "y," and "z") and the plane to display the projection.

In addition to the methods for 3D surface plotting, there are also straightforward generalizations of the line and scatter plot functions that are available for 2D axes, for example, plot, scatter, bar, and bar3d, which in the version that is available in the Axes3D class takes an additional argument for the *z* coordinates. Like their 2D relatives, these functions expect one-dimensional data arrays rather than the two-dimensional coordinate arrays used for surface plots.

When it comes to axes titles, labels, ticks, and tick labels, all the methods used for 2D graphs, as described earlier, are straightforwardly generalized to 3D graphs. For example, there are new methods set_zlabel, set_zticks, and set_zticklabels for manipulating the attributes of the new *z* axis. The Axes3D object also provides new class methods for 3D-specific actions and attributes. The view_init method can be used to change the angle from which the graph is viewed, and it takes the elevation and the azimuth, in degrees, as the first and second arguments.

Examples of how to use these 3D plotting functions are given in the following section, and the produced graphs are shown in Figure 4-25.

```
In [26]: fig, axes = plt.subplots(1, 3, figsize=(14, 4),
    ...:                          subplot_kw={'projection': '3d'})
    ...:
    ...: def title_and_labels(ax, title):
    ...:     ax.set_title(title)
    ...:     ax.set_xlabel("$x$", fontsize=16)
    ...:     ax.set_ylabel("$y$", fontsize=16)
    ...:     ax.set_zlabel("$z$", fontsize=16)
    ...:
    ...: x = y = np.linspace(-3, 3, 74)
    ...: X, Y = np.meshgrid(x, y)
    ...:
    ...: R = np.sqrt(X**2 + Y**2)
    ...: Z = np.sin(4 * R) / R
    ...:
    ...: norm = mpl.colors.Normalize(-abs(Z).max(), abs(Z).max())
    ...:
    ...: p = axes[0].plot_surface(
    ...:   X, Y, Z, rstride=1, cstride=1, linewidth=0, antialiased=False,
    ...:   norm=norm, cmap=mpl.cm.Blues)
    ...:
    ...: cb = fig.colorbar(p, ax=axes[0], shrink=0.6)
    ...: title_and_labels(axes[0], "plot_surface")
    ...:
    ...: p = axes[1].plot_wireframe(X, Y, Z, rstride=2, cstride=2,
    ...:                            color="darkgrey")
    ...: title_and_labels(axes[1], "plot_wireframe")
    ...:
    ...: cset = axes[2].contour(X, Y, Z, zdir='z', offset=0, norm=norm,
    ...:                        cmap=mpl.cm.Blues)
    ...: cset = axes[2].contour(X, Y, Z, zdir='y', offset=3, norm=norm,
    ...:                        cmap=mpl.cm.Blues)
    ...: title_and_labels(axes[2], "contour")
```

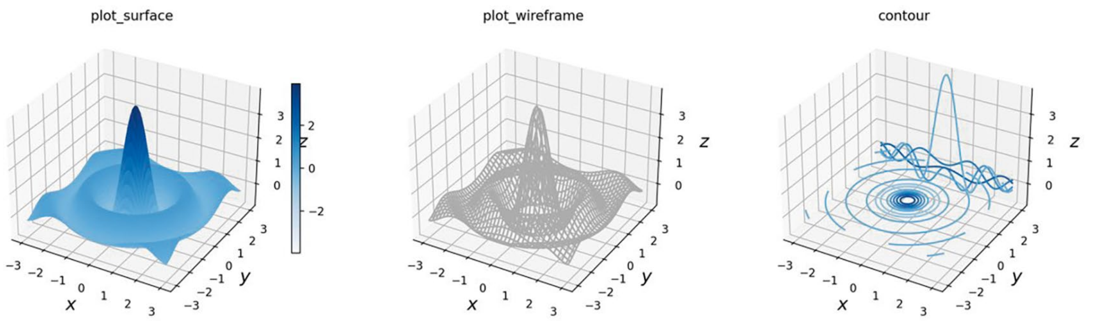


Figure 4-25. *3D surface and contour graphs generated by using* `plot_surface`*,* `plot_wireframe`*, and* `contour`

Summary

This chapter covered the basics of producing 2D and 3D graphics using Matplotlib. Visualization is one of the most important tools for computational scientists and engineers, both as an analysis tool while working on computational problems and for presenting and communicating computational results. Visualization is an integral part of the computational workflow, and it is equally important to be able to quickly visualize and explore data and to be able to produce picture-perfect publication-quality graphs, with detailed control over every graphical element. Matplotlib is a great general-purpose tool for both exploratory visualization and for producing publication-quality graphics. However, there are limitations to what can be achieved with Matplotlib, especially with respect to interactivity and high-quality 3D graphics. For more specialized cases, I therefore recommend exploring some of the other graphic libraries available in the scientific Python ecosystem, some of which were briefly mentioned at the beginning of this chapter.

Further Reading

Books dedicated to the Matplotlib library include *Matplotlib for Python Developers* by S. Tosi (Packt, 2009) and *Matplotlib Plotting Cookbook* by A. Devert (Packt, 2014). It is also widely covered in *Python Data Visualization Cookbook* by I. Milovanovi (Packt, 2013) and *Python for Data Analysis* by W. McKinney (O'Reilly, 2013). For interesting discussions on data visualization, style guides, and good practices in visualization, see *Visualize This* by N. Yau (Wiley, 2011) and *Beautiful Visualization* by J. Steele (O'Reilly, 2010).